2 Understanding ASP.NET Core

This chapter covers

- Why ASP.NET Core was created
- The many application paradigms of ASP.NET Core
- Approaches to migrating an existing application to ASP.NET Core

In this chapter, I provide some background on ASP.NET Core: why web frameworks are useful, why ASP.NET Core was created, and how to choose when to use ASP.NET Core. If you're new to .NET development, this chapter will help you understand the .NET landscape. If you're already a .NET developer, I provide guidance on whether now is the right time to consider moving your focus to .NET Core and .NET 7, as well as on the advantages ASP.NET Core can offer over previous versions of ASP.NET.

2.1 Using a web framework

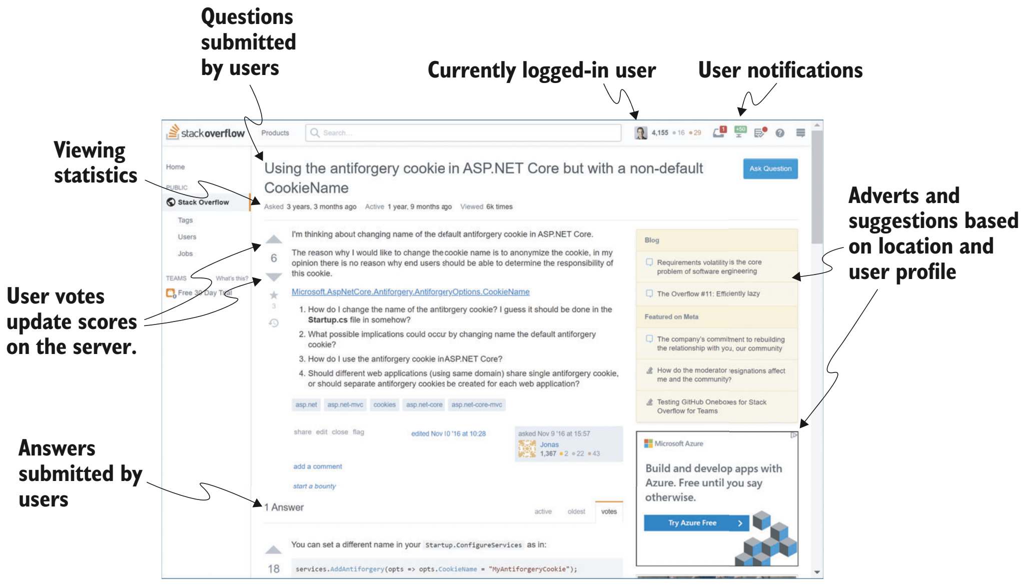
If you're new to web development, it can be daunting to move into an area with so many buzzwords and a plethora of ever-changing products. You may be wondering whether all those products are necessary. How hard can it be to return a file from a server?

Well, it's perfectly possible to build a static web application without the use of a web framework, but its capabilities will be limited. As soon as you want to provide any kind of security or dynamism, you'll likely run into difficulties, and the original simplicity that enticed you will fade before your eyes.

Just as desktop or mobile development frameworks can help you build native applications, ASP.NET Core makes writing web applications faster, easier, and more secure than trying to build everything from scratch. It contains libraries for common things like

- Creating dynamically changing web pages
- Letting users log in to your web app
- Letting users use their Facebook accounts to log in to your web app
- Providing a common structure for building maintainable applications
- Reading configuration files
- Serving image files
- Logging requests made to your web app

The key to any modern web application is the ability to generate dynamic web pages. A *dynamic web page* may display different data depending on the current logged-in user, or it could display content submitted by users. Without a dynamic framework, it wouldn't be possible to log in to websites or to display any sort of personalized data on a page. In short, websites like Amazon, eBay, and Stack Overflow (shown in figure 2.1) wouldn't be possible. Web frameworks for creating dynamic web pages are almost as old as the web itself, and Microsoft has created several over the years, so why create a new one?



Using the antiforgery cookie in ASP.NET Core but with a non-default CookieName

Asked 3 years, 3 months ago Active 1 year, 9 months ago Viewed 6k times

6

I'm thinking about changing name of the default antiforgery cookie in ASP.NET Core.

The reason why I would like to change the cookie name is to anonymize the cookie, in my opinion there is no reason why end users should be able to determine the responsibility of this cookie.

Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieName

1. How do I change the name of the antibrgery cookie? I guess it should be done in the **Startup.cs** file in somehow?
2. What possible implications could occur by changing name the default antiforgery cookie?
3. How do I use the antiforgery cookie inASP.NET Core?
4. Should different web applications (using same domain) share single antiforgery cookie, or should separate antiforgery cookies be created for each web application?

asp.net asp.net-mvc cookies asp.net-core asp.net-core-mvc

edited Nov 10 '16 at 10:28

asked Nov 9 '16 at 15:57
Jonas
1,367 •2 •22 •43

Blog
Requirements volatility is the core problem of software engineering
The Overflow #11: Efficiently lazy
Featured on Meta
The company's commitment to rebuilding the relationship with you, our community
How do the moderator resignations affect me and the community?
Testing GitHub Oneboxes for Stack Overflow for Teams

Build and develop apps with Azure. Free until you say otherwise.

1 Answer

18

You can set a different name in your `Startup.ConfigureServices` as in:

```
services.AddAntiforgery(opts => opts.CookieName = "MyAntiforgeryCookie");
```

Figure 2.1 The Stack Overflow website (https://stackoverflow.com) is built with ASP.NET and has almost entirely dynamic content.

2.2 Why ASP.NET Core was created

Microsoft's development of ASP.NET Core was motivated by the desire to create a web framework with five main goals:

- To be run and developed cross-platform
- To have a modular architecture for easier maintenance
- To be developed completely as open-source software
- To adhere to web standards
- To be applicable to current trends in web development, such as client-side applications and deployment to cloud environments

To achieve all these goals, Microsoft needed a platform that could provide underlying libraries for creating basic objects such as lists and dictionaries, and for performing tasks such as simple file operations. Up to this point, ASP.NET development had always been focused—and dependent—on the Windows-only .NET Framework. For ASP.NET Core, Microsoft created a lightweight platform that runs on Windows, Linux, and macOS called .NET Core (subsequently .NET), as shown in figure 2.2.

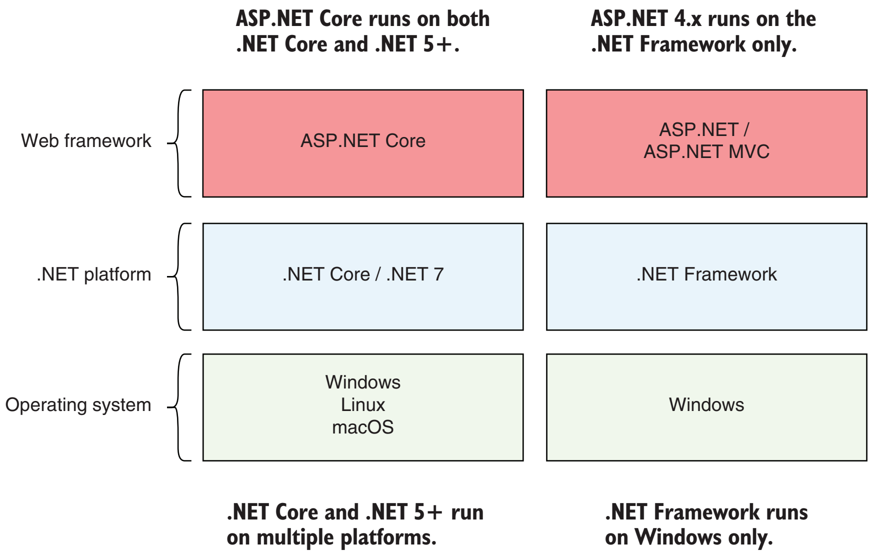

ASP.NET Core runs on both .NET Core and .NET 5+.
ASP.NET 4.x runs on the .NET Framework only.
Web framework
ASP.NET Core
ASP.NET / ASP.NET MVC
.NET platform
.NET Core / .NET 7
.NET Framework
Operating system
Windows Linux macOS
Windows
.NET Core and .NET 5+ run on multiple platforms.
.NET Framework runs on Windows only.

Figure 2.2 The relationships among ASP.NET Core, ASP.NET, .NET Core/.NET 5+, and .NET Framework. ASP.NET Core runs on .NET Core and .NET 5+, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so it's tied to the Windows OS.

DEFINITION .NET 5 was the next version of .NET Core after 3.1, followed by .NET 6 and .NET 7. It represents a unification of .NET Core and other .NET platforms in a single runtime and framework. It was considered to be the future of .NET, which is why Microsoft chose to drop the "Core" from its name. For consistency with Microsoft's language, I use the term *.NET 5+* to refer to .NET 5, .NET 6, and .NET 7, and the term *.NET Core* to refer to previous versions.

.NET Core (and its successor, .NET 5+) employs many of the same APIs as .NET Framework but is more modular. It implements a different set of features from those in .NET Framework, with the goal of providing a simpler programming model and modern APIs. It's a separate platform rather than a fork of .NET Framework, though it uses similar code for many of its APIs.

NOTE If you'd like to learn more about the .NET ecosystem, you can read two posts on my blog: "Understanding the .NET ecosystem: The evolution of .NET into .NET 7" (http://mng.bz/Ao0W) and "Understanding the .NET ecosystem: The introduction of .NET Standard" (http://mng.bz/ZqPZ).

The benefits and limitations of ASP.NET

ASP.NET Core is the latest evolution of Microsoft's popular ASP.NET web framework, released in June 2016. Previous versions of ASP.NET had many incremental updates, focusing on high developer productivity and prioritizing backward compatibility. ASP.NET Core bucks that trend by making significant architectural changes that rethink the way the web framework is designed and built.

ASP.NET Core owes a lot to its ASP.NET heritage, and many features have been carried forward from before, but ASP.NET Core is a new framework. The whole technology stack has been rewritten, including both the web framework and the underlying platform.

At the heart of the changes is the philosophy that ASP.NET should be able to hold its head high when measured against other modern frameworks, but existing .NET developers should continue to have a sense of familiarity.

To understand why Microsoft decided to build a new framework, it's important to understand the benefits and limitations of the legacy ASP.NET web framework.

The first version of ASP.NET was released in 2002 as part of .NET Framework 1.0. The ASP.NET Web Forms paradigm that it introduced differed significantly from the conventional scripting environments of classic ASP and PHP. ASP.NET Web Forms allowed developers to create web applications rapidly by using a graphical designer and a simple event model that mirrored desktop application-building techniques.

The ASP.NET framework allowed developers to create new applications quickly, but over time the web development ecosystem changed. It became apparent that ASP.NET Web Forms suffered from many problems, especially in building larger applications. In particular, a lack of testability, a complex stateful model, and limited influence on the generated HTML (making client-side development difficult) led developers to evaluate other options.

In response, Microsoft released the first version of ASP.NET MVC in 2009, based on the Model-View-Controller (MVC) pattern, a common web pattern used in frameworks such as Ruby on Rails, Django, and Java Spring. This framework allowed developers to separate UI elements from application logic, made testing easier, and provided tighter control of the HTML-generation process.

ASP.NET MVC has been through four more iterations since its first release, but all these iterations were built on the same underlying framework provided by the System .Web.dll file. This library is part of .NET Framework, so it comes preinstalled with all versions of Windows. It contains all the core code that ASP.NET uses when you build a web application.

This dependency brings both advantages and disadvantages. On one hand, the ASP.NET framework is a reliable, battle-tested platform that's fine for building web applications in Windows. It provides a wide range of features that have been in production for many years, and it's well known by virtually all Windows web developers.

On the other hand, this reliance is limiting. Changes to the underlying System.Web.dll file are far-reaching and, consequently, slow to roll out, which limits the extent to which ASP.NET is free to evolve and results in release cycles happening only every few years. There's also an explicit coupling with the Windows web host, Internet Information Services (IIS), which precludes its use on non-Windows platforms.

More recently, Microsoft declared .NET Framework to be "done." It won't be removed or replaced, but it also won't receive any new features. Consequently, ASP.NET based on System.Web.dll won't receive new features or updates either.

In recent years, many web developers have started looking at cross-platform web frameworks that can run on Windows as well as Linux and macOS. Microsoft felt the time had come to create a framework that was no longer tied to its Windows legacy; thus, ASP.NET Core was born.

With .NET 7, it's possible to build console applications that run cross-platform. Microsoft created ASP.NET Core to be an additional layer on top of console applications so that converting to a web application involves adding and composing libraries, as shown in figure 2.3.

When you add an ASP.NET Core web server to your .NET 7 app, your console application can run as a web application. ASP.NET Core contains a huge number of APIs, but you'll rarely need all the features available to you. Some of the features are built in and will appear in virtually every application you create, such as the ones for reading configuration files or performing logging. Other features are provided by separate libraries and built on top of these base capabilities to provide application-specific functionality, such as third-party logins via Facebook or Google.

Most of the libraries and APIs you'll use in ASP.NET Core are available on GitHub, in the Microsoft .NET organization repositories at https://github.com/dotnet/aspnetcore. You can find the core APIs there, including the authentication and logging APIs, as well as many peripheral libraries, such as the third-party authentication libraries.

All ASP.NET Core applications follow a similar design for basic configuration, but in general the framework is flexible, leaving you free to create your own code conventions. These common APIs, the extension libraries that build on them, and the design conventions they promote are covered by the somewhat-nebulous term ASP.NET Core.

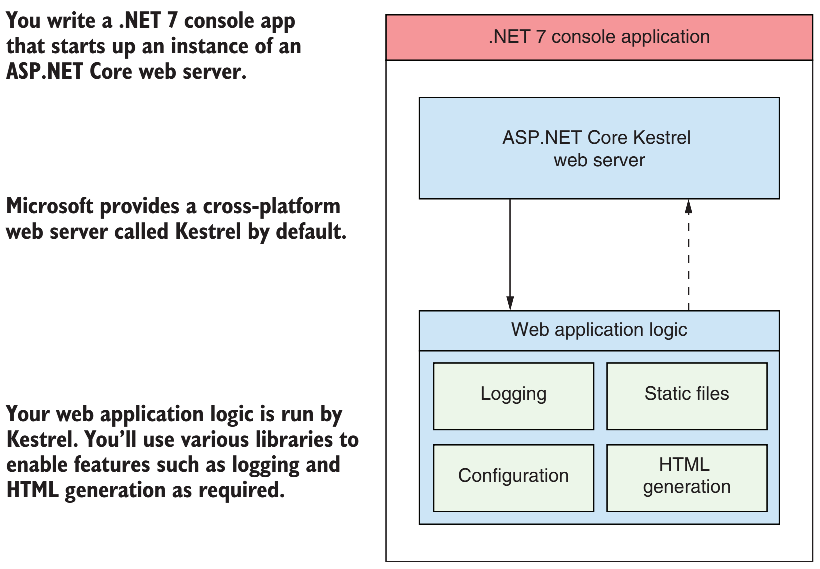

You write a .NET 7 console app that starts up an instance of an ASP.NET Core web server.
Microsoft provides a cross-platform web server called Kestrel by default.
Your web application logic is run by Kestrel. You'll use various libraries to enable features such as logging and HTML generation as required.
.NET 7 console application
ASP.NET Core Kestrel web server
Web application logic
Logging
Static files
Configuration
HTML generation

Figure 2.3 ASP.NET Core application model. The .NET 7 platform provides a base console application model for running command-line apps. Adding a web server library converts this model to an ASP.NET Core web app. You can add other features, such as configuration and logging, using various libraries.

2.3 Understanding the many paradigms of ASP.NET Core

In chapter 1 you learned that ASP.NET Core provides a generalized web framework that can be used to build a wide variety of applications. As you may recall from section 1.2, the main paradigms are

- *Minimal APIs*—Simple HTTP APIs that can be consumed by mobile applications or browser-based single-page applications (SPAs)
- *Web APIs*—An alternative approach for building HTTP APIs that adds more structure and features than minimal APIs
- *gRPC APIs*—Used to build efficient binary APIs for server-to-server communication using the gRPC protocol
- *Razor Pages*—Used to build page-based server-rendered applications
- *MVC controllers*—Similar to Razor Pages; used for server-based applications but without the page-based paradigm
- *Blazor WebAssembly*—A browser-based SPA framework using the WebAssembly standard, similar to JavaScript frameworks such as Angular, React, and Vue
- *Blazor Server*—Used to build stateful applications, rendered on the server, that send UI events and page updates over WebSockets to provide the feel of a client-side SPA but with the ease of development of a server-rendered application

All these paradigms use the core functionality of ASP.NET Core and layer the additional functionality on top. Each paradigm is suited to a different style of web application or API, so some may fit better than others, depending on what sort of application you're building.

Traditional page-based, server-side-rendered web applications are the bread and butter of ASP.NET development, both in the previous version of ASP.NET and now in ASP.NET Core. The Razor Pages and MVC controller paradigms provide two slightly different styles for building these types of applications but have many of the same concepts, as you'll see in part 2. These paradigms can be useful for building rich, dynamic websites, whether they're e-commerce sites, content management systems (CMSes), or large n-tier applications. Both the open-source CMS Orchard Core[1] (figure 2.4) and cloudscribe[2] CMS project, for example, are built with ASP.NET Core.

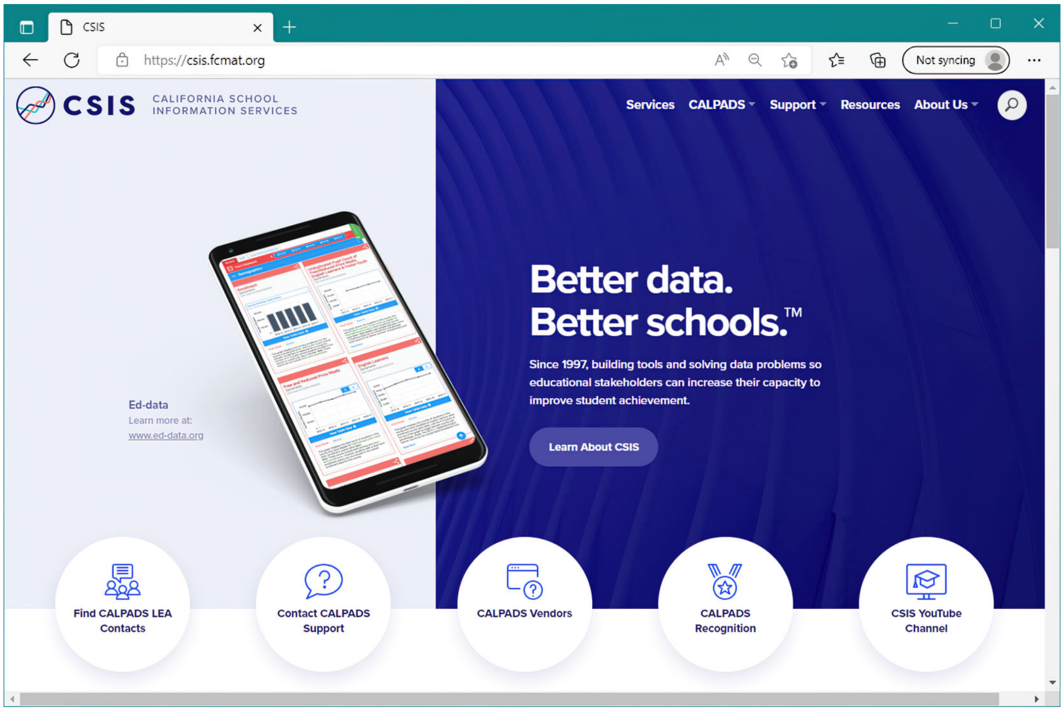

CSIS
https://csis.fcmat.org
Not syncing
CSIS CALIFORNIA SCHOOL INFORMATION SERVICES
Services
CALPADS
Support
Resources
About Us
Better data. Better schools.™
Since 1997, building tools and solving data problems so educational stakeholders can increase their capacity to improve student achievement.
Learn About CSIS
Ed-data
Learn more at:
www.ed-data.org
Find CALPADS LEA Contacts
Contact CALPADS Support
CALPADS Vendors
CALPADS Recognition
CSIS YouTube Channel

Figure 2.4 The California School Information Services website (https://csis.fcmat.org) is built with Orchard Core and ASP.NET Core.

In addition to server-rendered applications, ASP.NET core is ideally suited to building a REST or HTTP API server. Whether you're building a mobile app, a JavaScript SPA using Angular, React, Vue, or some other client-side framework, it's easy to create an ASP.NET Core application to act as the server-side API by using both the minimal API

[1] Orchard Core (https://orchardcore.net). Source code at https://github.com/OrchardCMS/OrchardCore.
[2] The cloudscribe project (https://www.cloudscribe.com). Source code at https://github.com/cloudscribe.

and web API paradigms built into ASP.NET Core. You'll learn about minimal APIs in part 1 and about web APIs in chapter 20.

DEFINITION *REST* stands for *representational state transfer*. RESTful applications typically use lightweight and stateless HTTP calls to read, post (create/update), and delete data.

ASP.NET Core isn't restricted to creating RESTful services. It's easy to create a web service or remote procedure call (RPC)-style service for your application, using gRPC for example, as shown in figure 2.5. In the simplest case, your application might expose only a single endpoint! ASP.NET Core is perfectly designed for building simple services, thanks to its cross-platform support and lightweight design.

DEFINITION gRPC is a modern open-source, high-performance RPC framework. You can read more at https://grpc.io.

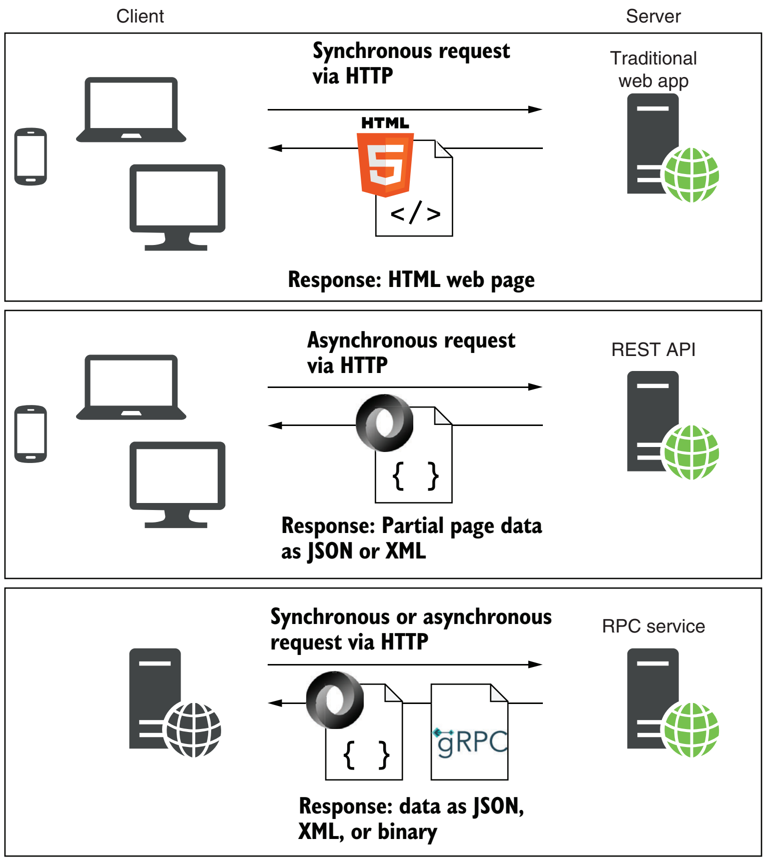

Client
Server
Synchronous request via HTTP
Traditional web app
HTML
</>
Response: HTML web page
Asynchronous request via HTTP
REST API
{ }
Response: Partial page data as JSON or XML
Synchronous or asynchronous request via HTTP
RPC service
{ }
gRPC
Response: data as JSON, XML, or binary

Figure 2.5 ASP.NET Core can act as the server-side application for a variety of clients: it can serve HTML pages for traditional web applications, act as a REST API for client-side SPA applications, or act as an ad hoc RPC service for client applications.

As well as server-rendered web apps, APIs, and gRPC endpoints, ASP.NET Core includes the Blazor framework, which can be used to build two very different styles of application. Blazor WebAssembly (WASM) apps run directly in your browser, in the same way as traditional JavaScript SPA frameworks such as Angular and React. Your .NET code is compiled to WebAssembly (https://webassembly.org) or executes on a .NET runtime compiled for WASM, and the browser downloads and runs it as it would a JavaScript app. This way you can build highly interactive client-side applications while using C# and all the .NET APIs and libraries you already know.

By contrast, Blazor Server applications run on the server. Each mouse click or keyboard event is sent to the server via WebSockets. Then the server calculates the changes that should be made to the UI and sends the required changes back to the client, which updates the page in the browser. The result is a "stateful" application that runs server-side but can be used to build highly interactive SPAs. The main downside of Blazor Server is that it requires a constant internet connection.

NOTE In this book I focus on building traditional page-based, server-side-rendered web applications and RESTful web APIs. I also show how to create background worker services in chapter 34. For more information on Blazor, I recommend *Blazor in Action,* by Chris Sainty (Manning, 2022).

With the ability to call on all these paradigms, you can use ASP.NET Core to build a wide variety of applications, but it's still worth considering whether ASP.NET Core is right for your specific application. That decision will likely be affected by both your experience with .NET and the application you want to build.

2.4 When to choose ASP.NET Core

In this section I'll describe some of the points to consider when deciding whether to use ASP.NET Core and .NET 7 instead of legacy .NET Framework ASP.NET. In most cases the decision will be to use ASP.NET Core, but you should consider some important caveats.

When choosing a platform, you should consider multiple factors, not all of which are technical. One such factor is the level of support you can expect to receive from its creators. For some organizations, limited support can be one of the main obstacles to adopting open-source software. Luckily, Microsoft has pledged to provide full support for Long Term Support (LTS) versions of .NET and ASP.NET Core for at least three years from the time of their release. And as all development takes place in the open, sometimes you can get answers to your questions from the general community as well as from Microsoft directly.

NOTE You can view Microsoft's official support policy at http://mng.bz/RxXP.

When deciding whether to use ASP.NET Core, you have two primary dimensions to consider: whether you're already a .NET developer and whether you're creating a new application or looking to convert an existing one.

2.4.1 If you're new to .NET development

If you're new to .NET development, you're joining at a great time! Many of the growing pains associated with a new framework have been worked out, and the result is a stable, high-performance, cross-platform application framework.

The primary language of .NET development, and of ASP.NET Core in particular, is C#. This language has a huge following, for good reason! As an object-oriented C-based language, it provides a sense of familiarity to those who are used to C, Java, and many other languages. In addition, it has many powerful features, such as Language Integrated Query (LINQ), closures, and asynchronous programming constructs. The C# language is also designed in the open on GitHub, as is Microsoft's C# compiler, code-named Roslyn (https://github.com/dotnet/roslyn).

NOTE I use C# throughout this book and will highlight some of the newer features it provides, but I won't be teaching the language from scratch. If you want to learn C#, I recommend *C# in Depth*, 4th ed., by Jon Skeet (Manning, 2019), and *Code Like a Pro in C#*, by Jort Rodenburg (Manning, 2021).

One big advantage of ASP.NET Core and .NET 7 over .NET Framework is that they enable you to develop and run on any platform. With .NET 7 you can build and run the same application on Mac, Windows, and Linux, and even deploy to the cloud using tiny container deployments.

Built with containers in mind

Traditionally, web applications were deployed directly to a server or, more recently, to a virtual machine. Virtual machines allow operating systems to be installed in a layer of virtual hardware, abstracting away the underlying hardware. This approach has several advantages over direct installation, such as easy maintenance, deployment, and recovery. Unfortunately, virtual machines are also heavy, in terms of both file size and resource use.

This is where containers come in. Containers are far more lightweight and don't have the overhead of virtual machines. They're built in a series of layers and don't require you to boot a new operating system when starting a new one, so they're quick to start and great for quick provisioning. Containers (Docker in particular) are quickly becoming the go-to platform for building large, scalable systems.

Containers have never been a particularly attractive option for ASP.NET applications, but with ASP.NET Core, .NET 7, and Docker for Windows, all that is changing. A lightweight ASP.NET Core application running on the cross-platform .NET 7 framework is perfect for thin container deployments. You can learn more about your deployment options in chapter 27.

In addition to running on each platform, one of the selling points of .NET is your ability to write and compile only once. Your application is compiled to Intermediate Language (IL) code, which is a platform-independent format. If a target system has the .NET 7 runtime installed, you can run compiled IL from any platform. You can develop on a Mac or a Windows machine, for example, and deploy *exactly the same files* to your production Linux machines. This compile-once, run-anywhere promise has finally been realized with ASP.NET Core and .NET 7.

> **TIP** You can go one step further and package the .NET runtime with your app in a so-called *self-contained deployment* (SCD). This way, you can deploy cross-platform, and the target machine doesn't even need .NET installed. With SCDs, the generated deployment files are customized for the target machine, so you're no longer deploying the same files everywhere in this case.

Many of the web frameworks available today use similar well-established *design patterns*, and ASP.NET Core is no different. Ruby on Rails, for example, is known for its use of the MVC pattern; Node.js is known for the way it processes requests using small discrete modules (called a *pipeline*); and dependency injection is available in a wide variety of frameworks. If these techniques are familiar to you, you should find it easy to transfer them to ASP.NET Core; if they're new to you, you can look forward to using industry best practices!

> **NOTE** *Design patterns* are solutions to common software design problems. You'll encounter a pipeline in chapter 4, dependency injection in chapters 8 and 9, and MVC in chapter 19.

Whether you're new to web development generally or only with .NET, ASP.NET Core provides a rich set of features with which you can build applications but doesn't overwhelm you with concepts, as the legacy ASP.NET framework did. On the other hand, if you're familiar with .NET, it's worth considering whether now is the time to take a look at ASP.NET Core.

2.4.2 If you're a .NET Framework developer creating a new application

If you're already a .NET Framework developer, you've likely been aware of .NET Core and ASP.NET Core, but perhaps you were wary about jumping in too soon or didn't want to hit the inevitable "version 1" problems. The good news is that ASP.NET Core and .NET are now mature, stable platforms, and it's absolutely time to consider using .NET 7 for your new apps.

As a .NET developer, if you aren't using any Windows-specific constructs such as the Registry, the ability to build and deploy cross-platform opens the possibility for cheaper Linux hosting in the cloud, or for developing natively in macOS without the need for a virtual machine.

.NET Core and .NET 7 are inherently cross-platform, but you can still use platform-specific features if you need to. Windows-specific features such as the Registry and Directory Services, for example, can be enabled with a Compatibility Pack that makes these APIs available in .NET 5+. They're available only when running .NET 5+ in Windows, not Linux or macOS, so you need to take care that such applications run only in a Windows environment or account for the potential missing APIs.

TIP The Windows Compatibility Pack is designed to help port code from .NET Framework to .NET Core/.NET 5+. See http://mng.bz/2DeX.

The hosting model for the previous ASP.NET framework was a relatively complex one, relying on Windows IIS to provide the web-server hosting. In a cross-platform environment, this kind of symbiotic relationship isn't possible, so an alternative hosting model has been adopted—one that separates web applications from the underlying host. This opportunity has led to the development of Kestrel, a fast, cross-platform HTTP server on which ASP.NET Core can run.

Instead of the previous design, whereby IIS calls into specific points of your application, ASP.NET Core applications are console applications that self-host a web server and handle requests directly, as shown in figure 2.6. This hosting model is conceptually much simpler and allows you to test and debug your applications from the command line, though it doesn't necessarily remove the need to run IIS (or the equivalent) in production.

ASP.NET Core and reverse proxies

You can expose ASP.NET Core applications directly to the internet so that Kestrel receives requests directly from the network. That approach is fully supported. It's more common, however, to use a reverse proxy between the raw network and your application. In Windows, the reverse-proxy server typically is IIS; in Linux or macOS, it might be NGINX, HAProxy, or Apache. There's even an ASP.NET Core-based reverse proxy library called YARP (https://microsoft.github.io/reverse-proxy) that you can use to build your own reverse proxy.

A *reverse proxy* is software responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy. This setup has several benefits, primarily security and performance for the web servers.

You may think that having a reverse proxy *and* a web server is somewhat redundant. Why not have one or the other? Well, one benefit is the decoupling of your application from the underlying operating system. The same ASP.NET Core web server, Kestrel, can be cross-platform and used behind a variety of proxies without putting any constraints on a particular implementation. Alternatively, if you wrote a new ASP.NET Core web server, you could use it in place of Kestrel without needing to change anything else about your application.

Another benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. Reverse proxies are often responsible for additional aspects, such as restarting a process that has crashed. Kestrel can remain a simple HTTP server, not having to worry about these extra features, when it's used behind a reverse proxy. You can think of this approach as being a simple separation of concerns: Kestrel is concerned with generating HTTP responses, whereas the reverse proxy is concerned with handling the connection to the internet.

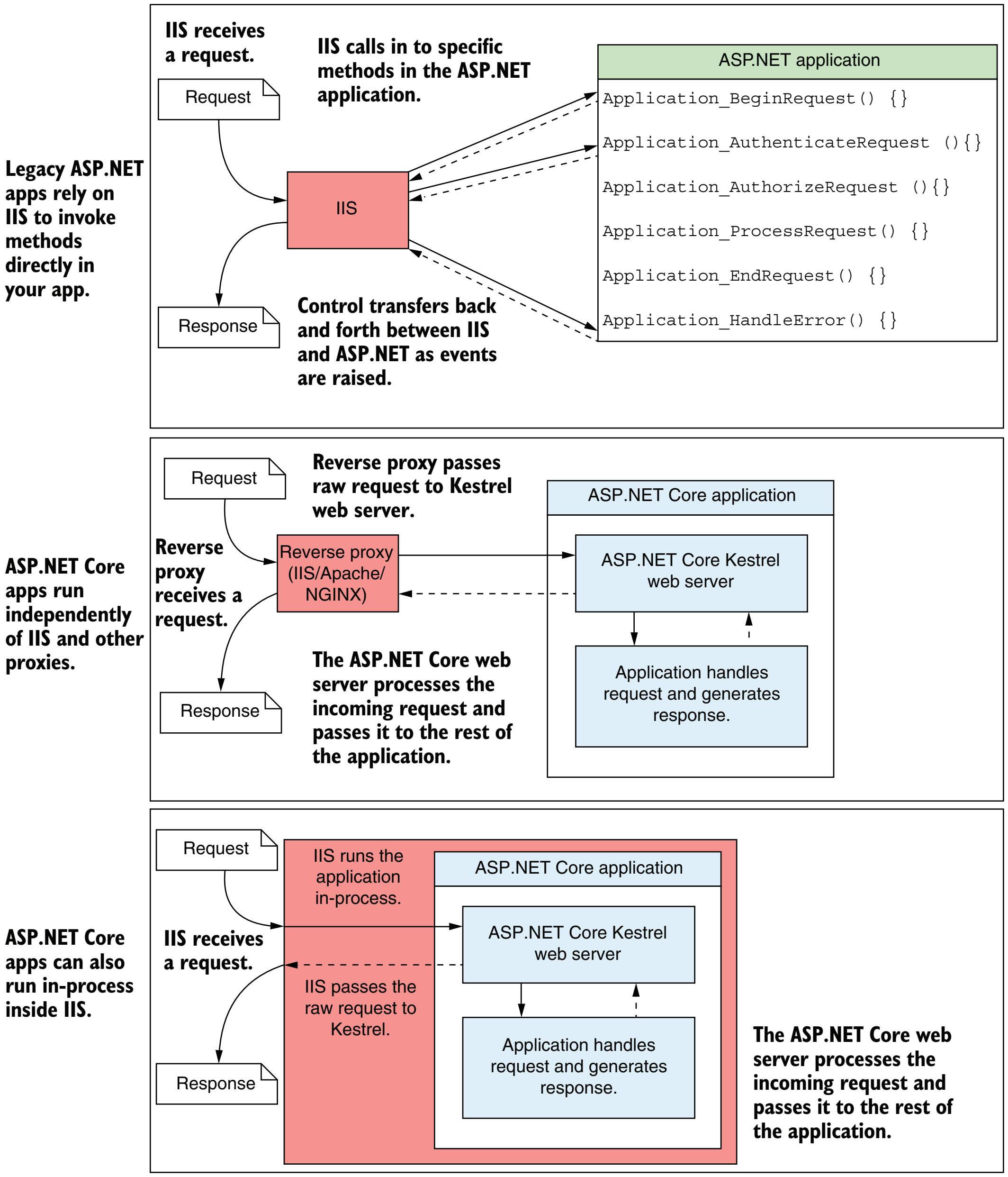

IIS receives a request.
IIS calls in to specific methods in the ASP.NET application.
ASP.NET application
Request
Application_BeginRequest() {}
Application_AuthenticateRequest (){}
Application_AuthorizeRequest (){}
Application_ProcessRequest() {}
Application_EndRequest() {}
Application_HandleError() {}
Legacy ASP.NET apps rely on IIS to invoke methods directly in your app.
IIS
Response
Control transfers back and forth between IIS and ASP.NET as events are raised.
Request
Reverse proxy passes raw request to Kestrel web server.
ASP.NET Core application
Reverse proxy receives a request.
Reverse proxy (IIS/Apache/ NGINX)
ASP.NET Core Kestrel web server
ASP.NET Core apps run independently of IIS and other proxies.
Application handles request and generates response.
Response
The ASP.NET Core web server processes the incoming request and passes it to the rest of the application.
Request
IIS runs the application in-process.
ASP.NET Core application
ASP.NET Core Kestrel web server
ASP.NET Core apps can also run in-process inside IIS.
IIS receives a request.
IIS passes the raw request to Kestrel.
Application handles request and generates response.
Response
The ASP.NET Core web server processes the incoming request and passes it to the rest of the application.

Figure 2.6 The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). In the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response but has no deeper knowledge of the application.

NOTE By default, when running in Windows, ASP.NET Core runs *inside* IIS, as shown in figure 2.6, which can provide better performance than the reverse-proxy version. This is primarily a deployment detail and doesn't change the way you build ASP.NET Core applications.

Changing the hosting model to use a built-in HTTP web server has created another opportunity. Performance has been something of a sore point for ASP.NET applications in the past. It's certainly possible to build high-performing applications—Stack Overflow (https://stackoverflow.com) is a testament to that fact—but the web framework itself isn't designed with performance as a priority, so it can end up being an obstacle.

To make the product competitive cross-platform, the ASP.NET team focused on making the Kestrel HTTP server as fast as possible. TechEmpower (https://www.techempower.com/benchmarks) has been running benchmarks on a wide range of web frameworks from various languages for several years now. In round 20 of the plain-text benchmarks, TechEmpower announced that ASP.NET Core with Kestrel was among the 10 fastest of more than 400 frameworks tested![3]

Web servers: Naming things is hard

One difficult aspect of programming for the web is the confusing array of often-conflicting terminology. If you've used IIS, for example, you may have described it as a web server or possibly a web host. Conversely, if you've ever built an application with Node.js, you may have also referred to that application as a web server. Or you may have called the physical machine on which your application runs a web server. Similarly, you may have built an application for the internet and called it a website or a web application, probably somewhat arbitrarily based on the level of dynamism it displayed.

In this book, when I say *web server* in the context of ASP.NET Core, I'm referring to the HTTP server that runs as part of your ASP.NET Core application. By default, this server is the Kestrel web server, but that's not a requirement. It's possible to write a replacement web server for Kestrel if you so desire.

The web server is responsible for receiving HTTP requests and generating responses. In the previous version of ASP.NET, IIS took this role, but in ASP.NET Core, Kestrel is the web server.

I'll use the term *web application* in this book to describe ASP.NET Core applications, regardless of whether they contain only static content or are dynamic. Either way, these applications are accessed via the web, so that name seems to be the most appropriate.

Many of the performance improvements made to Kestrel came not from the ASP.NET team members themselves, but from contributors to the open-source project on GitHub (https://github.com/dotnet/aspnetcore). Developing in the open means that you typically see fixes and features make their way to production faster than you would

[3] As always in web development, technology is in a constant state of flux, so these benchmarks will evolve over time. Although ASP.NET Core may not maintain its top-10 slot, you can be sure that performance is one of the key focal points of the ASP.NET Core team.

for the previous version of ASP.NET, which was dependent on .NET Framework and Windows and, as such, had long release cycles.

By contrast, .NET 5+ and hence ASP.NET Core are designed to be released in small increments. Major versions will be released on a predictable cadence, with a new version every year and a new LTS version released every two years (http://mng.bz/1qrg). In addition, bug fixes and minor updates can be released as and when they're needed. Additional functionality is provided in NuGet packages independent of the underlying .NET 5+ platform.

> **NOTE** NuGet is a package manager for .NET that enables you to import libraries into your projects. It's equivalent to Ruby Gems, npm for JavaScript, or Maven for Java.

To enable this approach to releases, ASP.NET Core is highly modular, with as little coupling to other features as possible. This modularity lends itself to a pay-for-play approach to dependencies, where you start with a bare-bones application and add only the libraries you require, as opposed to the kitchen-sink approach of previous ASP.NET applications. Even MVC is an optional package! But don't worry—this approach doesn't mean that ASP.NET Core is lacking in features, only that you need to opt into them. Some of the key infrastructure improvements include

- Middleware pipeline for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (web API) infrastructure
- Highly extensible configuration system
- Standardized, extensible logging system
- Uses asynchronous programming by default for built-in scalability on cloud platforms

Each of these features was possible in the previous version of ASP.NET but required a fair amount of additional work to set up. With ASP.NET Core, they're all there, ready and waiting to be connected.

Microsoft fully supports ASP.NET Core, so if you want to build a new system, there's no significant reason not to use it. The largest obstacle you're likely to come across is wanting to use programming models that are no longer supported in ASP.NET Core, such as Web Forms or WCF Server, as I'll discuss in the next section.

I hope that this section whetted your appetite to use ASP.NET Core for building new applications. But if you're an existing ASP.NET developer considering whether to convert an existing ASP.NET application to ASP.NET Core, that's another question entirely.

2.4.3 Converting an existing ASP.NET application to ASP.NET Core

By contrast with new applications, an existing application presumably already provides value, so there should always be a tangible benefit to performing what may amount to a significant rewrite in converting from ASP.NET to ASP.NET Core. The advantages of adopting ASP.NET Core are much the same as those for new applications: cross-platform deployment, modular features, and a focus on performance. Whether the benefits are sufficient will depend largely on the particulars of your application, but some characteristics make conversion more difficult:

- Your application uses ASP.NET Web Forms.
- Your application is built with WCF.
- Your application is large, with many advanced MVC features.

If you have an ASP.NET Web Forms application, attempting to convert it directly to ASP.NET Core isn't advisable. Web Forms is inextricably tied to System.Web.dll and IIS, so it will likely never be available in ASP.NET Core. Converting an application to ASP.NET Core effectively involves rewriting the application from scratch, not only shifting frameworks, but also potentially shifting design paradigms.

All is not lost, however. Blazor server provides a stateful, component-based application that's *similar* to the Web Forms application model. You may be able to gradually migrate your Web Forms application page by page to an ASP.NET Core Blazor server application.[4] Alternatively, you could slowly introduce web API concepts into your Web Forms application, reducing the reliance on legacy Web Forms constructs such as ViewState, with the goal of ultimately moving to an ASP.NET Core web API application.

Windows Communication Foundation (WCF) is only partially supported in ASP.NET Core. It's possible to build client-side WCF services using the libraries provided by ASP.NET Core (https://github.com/dotnet/wcf) and to build server-side WCF services by using the Microsoft-supported community-driven project CoreWCF.[5] These libraries don't support all the APIs available in .NET Framework WCF (distributed transactions and some message security formats, for example), so if you absolutely need those APIs, it may be best to avoid ASP.NET Core for now.

TIP If you like WCF's contract-based RPC-style of programming but don't have a hard requirement for WCF itself, consider using gRPC instead. gRPC is a modern RPC framework with many concepts that are similar to WCF, and it's supported by ASP.NET Core out of the box (http://mng.bz/wv9Q).

If your existing application is complex and makes extensive use of the previous MVC or web API extensibility points or message handlers, porting your application to ASP.NET Core may be more difficult. ASP.NET Core is built with many features similar to the previous version of ASP.NET MVC, but the underlying architecture is different. Several of the previous features don't have direct replacements, so they'll require rethinking.

The larger the application is, the greater the difficulty you're likely to have converting your application to ASP.NET Core. Microsoft itself suggests that porting an application from ASP.NET MVC to ASP.NET Core is at least as big a rewrite as porting

[4] There is a community-driven effort to create Blazor versions of common WebForms components (http://mng.bz/PzPP). Also see an e-book for Blazor for Web Forms developers at http://mng.bz/JgDv.

[5] You can find the CoreWCF libraries at https://github.com/corewcf/corewcf and details on upgrading a WCF service to .NET 5+ at http://mng.bz/mVg2.

from ASP.NET Web Forms to ASP.NET MVC. If that suggestion doesn't scare you, nothing will!

If an application is rarely used, isn't part of your core business, or won't need significant development in the near term, I suggest that you *don't* try to convert it to ASP.NET Core. Microsoft will support .NET Framework for the foreseeable future (Windows itself depends on it!), and the payoff in converting these fringe applications is unlikely to be worth the effort.

So when *should* you port an application to ASP.NET Core? As I've already mentioned, the best opportunity to get started is on small new greenfield projects instead of existing applications. That said, if the existing application in question is small or will need significant future development, porting may be a good option.

It's always best to work in small iterations if possible when porting an application, rather than attempt to convert the entire application at the same time. Luckily, Microsoft provides tools for that purpose. A set of System.Web adapters, a .NET-based reverse proxy called YARP (Yet Another Reverse Proxy; http://mng.bz/qr92), and tooling built into Visual Studio can help you implement the strangler fig pattern (http://mng.bz/rW6J). This tooling allows you to migrate your application one page/API at a time, reducing the risk associated with porting an ASP.NET application to ASP.NET Core.

In this chapter, we walked through some of the historical context of ASP.NET Core, as well as some of the advantages of adopting it. In chapter 3, you'll create your first application from a template and run it. We'll walk through each of the main components that make up your application and see how they work together to render a web page.

Summary

- Web frameworks provide a way to build dynamic web applications easily.
- ASP.NET Core is a web framework built with modern software architecture practices and modularization as its focus.
- ASP.NET Core runs on the cross-platform .NET 7 platform. You can access Windows-specific features such as the Windows Registry by using the Windows Compatibility Pack.
- .NET 5, .NET 6, and .NET 7 are the next versions of .NET Core after .NET Core 3.1.
- ASP.NET Core is best used for new greenfield projects.
- Legacy technologies such as WCF Server and Web Forms can't be used directly with ASP.NET Core, but they have analogues and supporting libraries that can help with porting ASP.NET applications to ASP.NET Core.
- You can convert an existing ASP.NET application to ASP.NET Core gradually by using the strangler fig pattern, using tooling and libraries provided by Microsoft.
- ASP.NET Core apps are often protected from the internet by a reverse-proxy server, which forwards requests to the application.